1991-06

Complementary metal oxide silicon cyclic redundancy check generators.

Chin, Miao

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

COMPLEMENTARY METAL OXIDE SILICON CYCLIC REDUNDANCY CHECK GENERATORS

by

Chin, Miao

June 1991

Thesis Advisor: Chyan Yang

Approved for public release; distribution is unlimited.

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification: Unclassified	1b Restrictive Markings
2a Security Classification Authority	3 Distribution Availability of Report: Approved for public release; distribution is unlimited.
2b Declassification/Downgrading Schedule	
4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
6a Name of Performing Organization: Naval Postgraduate School; 6b Office Symbol (*If Applicable*): EC	7a Name of Monitoring Organization: Naval Postgraduate School
6c Address (*city, state, and ZIP code*): Monterey, CA 93943-5000	7b Address (*city, state, and ZIP code*): Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization; 8b Office Symbol (*If Applicable*)	9 Procurement Instrument Identification Number
8c Address (*city, state, and ZIP code*)	10 Source of Funding Numbers: Program Element Number; Project No; Task No; Work Unit Accession No

11 Title (*Include Security Classification*)
COMPLEMENTARY METAL OXIDE SILICON CYCLIC REDUNDANCY CHECK GENERATORS

12 Personal Author(s)
Chin, Miao

13a Type of Report	13b Time Covered	14 Date of Report (*year, month,day*)	15 Page Count
Master's Thesis	From To	June1991	71

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms (*continue on reverse if necessary and identify by block number*)
Field	Group	Subgroup	Cyclic Redundancy Check; Linear Feedback Shift Register; Built-in Self Test; Design for Testability; Level Sensitive Scan Design; Circuit Under Test; Weighted Pseudorandom Pattern Generator; Very Large Scale Integrated Circuits, VLSI.

19 Abstract (*continue on reverse if necessary and identify by block number*)

This thesis introduces an economical way of implementing the test pattern generation for built-in test. A layout generator as well as a netlist generator are written and validated. In addition, we use the netlist generator to investigate the properties of nonprimitive polynomials.

20 Distribution/Availability of Abstract	21 Abstract Security Classification
[X] unclassified/unlimited [ ] same as report [ ] DTIC users	Unclassified

22a Name of Responsible Individual	22b Telephone (*Include Area code*)	22c Office Symbol
Yang, Chyan	(408) 646-2266	EC/Ya

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted

All other editions are obsolete

security classification of this page

Unclassified

Approved for public release; distribution is unlimited.

Complementary Metal Oxide Silicon Cyclic Redundancy Check Generators

by

Chin, Miao
Commander, Republic of China Navy
B.S., Chinese Naval Academy, 1978

Submitted in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE IN ENGINEERING SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1991

Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

This thesis introduces an economical way of implementing the test pattern generation for built-in test. A layout generator as well as a netlist generator are written and validated. In addition, we use the netlist generator to investigate the properties of nonprimitive polynomials.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisors Professor Chyan Yang and Professor Herschel H. Loomis for their guidance and assistance. And a very special thank you to my parents and my wife Su, Miao-Ying for her love and support which allowed me to successfully complete my graduate education.

I. INTRODUCTION

When a logic chip or board has been fabricated, testing is performed on the finished product to ensure that it is free of manufacturing defects. The device is mounted in the test socket of a tester which drives its input terminals with logic signals called test vectors. The responses to these input stimuli are then obtained at the output terminals of the device. The test patterns are generated either by a software algorithm or by hardware built into the test equipment or by hardware actually embedded in the device to be tested. Good responses are obtained by simulation or by measurement of the output of a number of "good" devices. Responses to the test patterns may be compressed into a unique binary number which accumulates all the response data. The simplest compressed measure is the parity bit used in digital communication. Responses then are compared with "good" outputs to verify correct operation of the device under test.

At low levels of integration, it is possible for an engineer to manually write the test patterns from the functional specification for the device. However, this can be a difficult and time consuming process for devices containing just a few hundred circuits, and manual test pattern generation for Very Large Scale Integrated (VLSI) devices may require many engineer-years of effort. The purpose of this thesis is to investigate the on-chip test pattern generation so that we can avoid the time and storage required when we do it off-chip. An economical way of implementing the test pattern is introduced. A layout generator as well as a netlist generator are written and

validated. In addition, we use netlist generator to investigate the properties of nonprimitive polynomials.

The organization of this thesis is described as follows. In Chapter Two, we discuss Built-in Self Test (BIST). Chapter Three discusses Cyclic Redundancy Code (CRC) dividers and Linear Feedback Shift Registers (LFSR) and explains how these devices can be used as pseudorandom pattern generator. In Chapter Four, the weighted pseudorandom pattern generator (WPRPG) is discussed and a set of experiments are performed to investigate the behavior of WPRPG. Chapter Five summarizes the results and presents the conclusions of the thesis. "C" program listings for the thesis project are listed in Appendix A.

II. BUILT-IN SELF TEST

A. DESIGN FOR TESTABILITY

Design for testability (DFT) is motivated by the need to reduce the costs of testing. The main testability considerations are difficulty of test generation, test sequence length, test application cost, fault coverage and fault resolution. The costs are basically those of the computer time required to generate test patterns, personnel to write the test pattern program and test equipment [Ref.1]. The objective of DFT is to design circuits from the outset in order to limit in magnitude the test generation efforts and test verification [Ref.2].

Three basic approaches for designing are ad-hoc testing, structured design approach, and built-in self testing. Here we focus only on built-in self test approach using built-in logic block observation (BILBO) technique. This technique, uses the scan path and level sensitive scan design (LSSD) concept combines it with the signature analysis concept [Ref.3]. The complete test generation and observation arrangement can be implemented as shown in Fig. 2.1 [Ref.2]. The pattern generator on the left is used as the sequence generator to generate test patterns, which will be applied to the combinational circuitry under test. The register on the right is used as response analyzer. After a certain number of pattens applied, the responses collected by the analyzer will be used to determine if the circuit is correct. In LSSD or other structural design techniques, a considerable amount of test data volume is involved with the shifting in and out. Testing a circuit requires the application of the test patterns and the comparison of the actual circuit

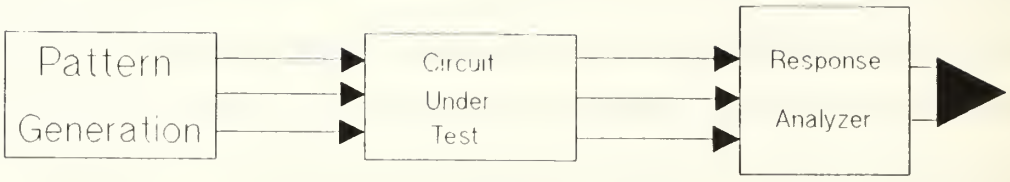

Figure 2.1 The Complete Test Generation

response with the correct response. General-purpose testers, though commonly used for this purpose, are very expensive. Tester cost is not the only difficulty encountered in using an external tester. There are also problems:

- *Time* - The turnaround time to generated test patterns, the time taken to apply the test patterns, and the computation time are growing too large.
- *Volume* - The number of test patterns is becoming too large to be handled efficiently by the tester hardware.

B. BUILT-IN TEST

Several techniques have been proposed for reducing the complexity of external testing by moving some of the tester functions onto the chip itself or

onto the board on which the chip are mounted. One approach to eliminate testing problem above is to incorporated built-in self test circuitry into the circuit under test (CUT). In this technique, test patterns and output responses are generating in BIST designs. The test patterns are fed as input to the circuit under test, and output responses from the circuit under test are then compressed through signature analyzer to form a signature. In the next section we will discuss test pattern generation in detail.

To solve the problem of analyzing and storing the large amount of data, which is required for a good response generation, we may use a compact testing method. This technique, signature analysis, reduces the memory and additional circuit units [Ref.4]. This technique can be used when it is not feasible to compare test result data with reference data, for each input test pattern. If the reference data available at the same rate and in synchronism with the data being tested there is no advantage using this technique.

By analyzing the signature, we can determine whether the circuit under test is faulty or not. Since compact testing compresses response data and compares the signature only once, the difficulty of analysis and storage of huge amounts of test response data can be avoided. Furthermore, BIST has another important advantage that the circuit under test is fed with random test patterns at the functional clock rate. Hence, it is possible to perform high speed testing using internal test device built in the circuit under test. For a detailed discussion on signature analysis one can consult Barus' thesis [Ref.5].

C. TEST GENERATION

In designing a circuit one always wants to have a way to verify the design before and after the chip is fabricated. Before the fabrication one can have design verification by circuit simulation either at functional, switch, or device level. After the chip is fabricated, a designer has to find a way of testing the circuit by providing various test patterns (controllability) to the circuit and measure the output (observability). In practice, a designer assumes that the faults in a circuit can be either stuck-at-0 or stuck-at-1. A node stuck-at-0 means the node is shorted to the ground, while a node stuck-at-1 means the node is shorted to Vdd or power supply. Although there are other fault models the stuck-at fault is the most commonly used one. Moreover, to simplify the problem, the 'single' fault model is used: a circuit is faulty only at a single node. This simplification may seem unreasonable but practically it is useful and is used in today's theoretical research and industrial circuits [Ref.6]. A network of n nodes has 2n possible single faults since for each node there are two possible faults: either stuck-at-0 or stuck-at-1. To exhaustively test the circuit we may have to generate all possible test patterns and apply them to the primary inputs of the circuit. The circuit complexity may be characterize by the number of primary inputs. The number of internal nodes is generally more than square of the number of inputs.

Consider a 2-input 'and' gate with nodes A, B as primary inputs and node C as the primary output. Possible exhaustive test patterns are AB = { 00, 01, 10, 11 }. For node B with stuck-at-0 fault to be detected, the test pattern must be 11. Other faults and their corresponding detecting vectors are shown in Table

2.1. To have a 100% fault coverage, we need to apply all four possible test patterns for the AND gate. For a circuit of k inputs, the number of exhaustive input patterns would be 2^k. Generation of test patterns off-line requires tremendous storage and extremely long data transmission time. To save both space and time the 'random' test patterns should be supplied on chip hence the built-in test pattern generation. The randomness is not a necessary condition for the built-in test; all we need is to provide a set of test patterns that can capture all faults. The naive approach would be an k-bit counter that can generate (or count) patterns from 0 up to $2^k - 1$. However, the counter approach is more expensive than the linear feedback shift register (LFSR) to be explained in the next chapter because sufficient coverage may be achived by test patterns generated from an LFSR.

TABLE 2.1 THE FAULTS AND THEIR CORRESPONDING DETECTING VECTOR

Fault-type	Detecting vector
$A\ s-a-0$	(1 1)
$A\ s-a-1$	(0 1)
$B\ s-a-0$	(1 1)
$B\ s-a-1$	(1 0)
$C\ s-a-0$	(1 1)
$C\ s-a-1$	(0 0)

Test generation is used to search a sequence of input test vectors which verify the correctness of the circuit and test verification is concerned to find measures of effectiveness of a given set of test vectors. All of these consideration should be considered when designing a test strategy and test plan. Moreover, it is sometimes important to locate the fault as well as detecting it. The strategy of testing can be changed depends on whether it is desired to detect the fault only, or to detect and to locate the fault. The manual generation of test patterns is a difficult, time consuming job even for moderate circuits. Using of pseudo-random pattern generator (discussed in Chapter III) and generation of a test vector which detects a single failure indicates the degree of underestimation of the importance of the test generation process. Faults simulation has been the goal of the test generation, yielding a quantitative measure of test effectiveness. In other words, a test sequence is considered good if it can detect a high percentage of the possible faults in the circuit under test.

Sometimes the designer has the knowledge about the circuit behavior and does not need all exhaustive test patterns for capturing faults. In these cases, the test patterns are not uniformly distributed. The best approach to save unnecessary pattern generation time is to provide test patterns that reflect the circuit behavior. Exhaustive pattern generation may be technically feasible and can cover all faults but it is not economical feasible for circuits with large number of primary inputs. A circuit with 64 inputs is common in today's digital design and exhaustive testing patterns of $2^{64} = 1.84467 \times 10^{19}$ or about 10^{20}. With a testing device of 20 MHz, test pattern generation frequency (or 50 ns per pattern) we need $10^{20} \times 50ns = 10^{12}$ sec. or 10^4 *centuries*. For

example, if we know a certain bit would be '1' 75% of the test patterns we then can 'bias' our test patterns to force '1' at that node 75% of the time.

To increase the efficiency of random test pattern generation, Parker [Ref. 7] proposed a method by adaptively controlling the source statistics. Adding an adaptive mechanism to random test pattern generation adds a trivial amount of additional computation to the process. Based on method mentioned above, in [Ref. 8] the new test generation method named Weighted Test Pattern Generation (discussed in Chapter IV) has been proposed to reduce the number of test pattens required for high fault coverage. Using this method, the adaptive mechanism is provided by monitoring the logic-level transition activity. In [Ref. 9] the adaptive weighted test pattern generation is applied to develop a minimal set of test patterns with maximum fault coverage. The result shows that adaptive test generation of a set of probabilistic stimuli detect a significantly large percentage of the fault in the selected circuits. The above technique indicates that a method for generating weighted test patterns is a powerful way to reduce the number of test patterns. The generation of weighted test patterns can be achieved by weighted LFSR which is discussed in Chapter IV.

III. CRC AND PRPG GENERATORS

A. OVERVIEW OF CYCLIC REDUNDANCY CODE (CRC)

CRC is commonly used in today's digital communication systems for error detection. For example, CRC-4 is used in European ISDN primary rate access and CRC-12 is used in U-interface superframe of the ISDN basic rate access [Ref. 10]. A CRC-d code is an n-d linear code that consists of n-bits of information and d bits of checkword. A CRC can be implemented in hardware by an LFSR and is called a CRC divider, to be explained in the next section. A message of n-bit is divided by a CRC-d divider and the d-bit remainder becomes a checkword that appends to the message bits. A transmitter sends this $n + d$ bit block to the receiver. A receiver applies the received n-bit message to its CRC divider and the remainder or checkword is then compared with the received checkword. If the checkword generated at receiver end is the same as the checkword received, the transmission is treated as error free; otherwise transmission error is detected. Each CRC-d code is defined by a characteristic polynomial p(x) and each p(x) corresponds to a hardware implementation called CRC divider.

An example is instructive to understand the CRC process. Let the 10-bit message be 1101011011 and the CRC divider polynomial $g(x) = x^4 + x + 1$. The message can be expressed as $m(x) = x^9 + x^8 + x^6 + x^4 + x^3 + x + 1$. Since $m(x) = (x^3 + x^2 + x) \bmod g(x) = r(x) \bmod g(x)$, the remainder in the CRC-4 divider is 1110, and the transmitter sends out the concatenation of m(x) and r(x) or 11010110111110 to the receiver. When the receiver receives the message, the

first ten message bits will be applied to its CRC-4 divider which produces its own remainder. If the transmission is error free then there would be no error in the message bits and the remainder generated by the receiver should be the same as that it has received i,e., 1110. Suppose, we have a unreliable transmission and the message bits received by the receiver is 1001011011 or $m'(x) = x^9 + x^6 + x^4 + x^3 + x + 1$. Now the remainder generated by the receiver would be 00001 and is different from 1110 so that the transmission error is detected.

B. CRC DIVIDER

The general configuration of a CRC divider is a set of delay flip-flops and XOR gates connected in such a way that the last stage output feeds to a set of intermediate stages. Fig. 3.1 shows an example of CRC-9 divider. Note that a CRC-d divider requires d flip-flops and k XOR gates for the p(x) with k terms. It has been shown in [Ref. 11] that when the generator polynomial is primitive the number of distinct patterns a CRC divider can produce would be 2^k for a p(x) with k terms, assuming the initial value in the divider is zero. Table 3.1 shows example of various patterns generated by non-primitive $p_1(x) = x^5 + x^4 + x^3 + x^2 + x + 1$ and primitive $p_2(x) = x^5 + x^2 + 1$ polynomials. Notice that $p_1(x)$ and $p_2(x)$ are initialized to the same value of 10011. The primitive $p_2(x)$ cycles through all possible $2^5 - 1$ or 31 patterns except 0. The nonprimitive polynomial $p_1(x)$, can only cycle through 6 distinct patterns. In other words, if we were to use $p_1(x)$ for pattern generation we would miss 31 – 6 = 25 patterns hence reduce the fault coverage.

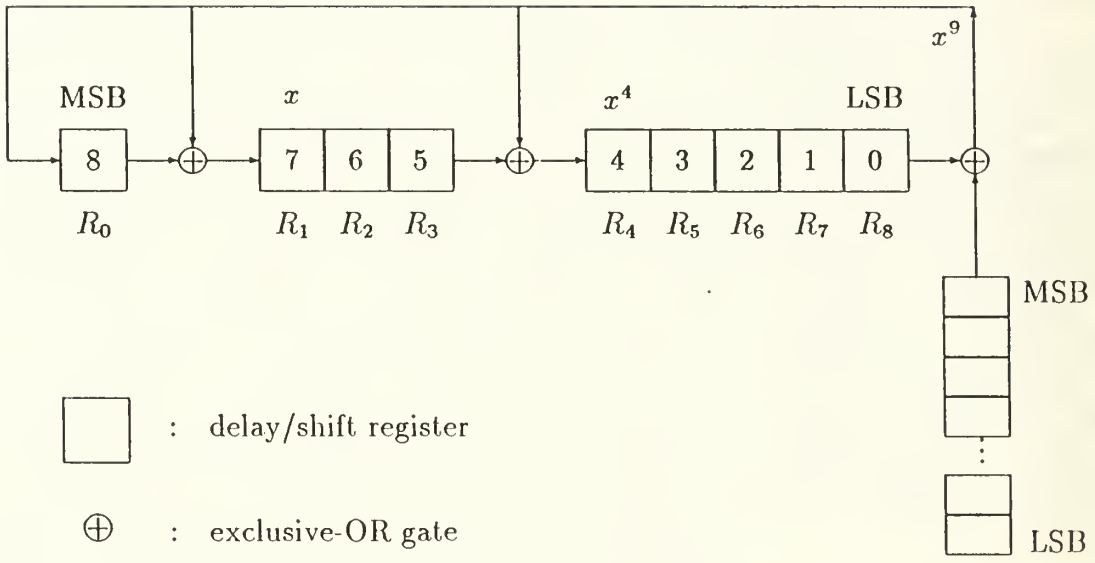

Figure 3.1 Method for Obtaining CRC-9 Checkwords Using a Shift Register

TABLE 3.1 VARIOUS PATTERNS GENERATED BY NONPRIMITIVE AND PRIMITIVE

Number of Test Pattern	Primitive Test Pattern	Nonprimitive Test Pattern
1	10011	10011
2	00011	11001
3	00110	01101
4	01100	11010
5	11000	01011
6	10101	10110
7	01111	10011*
8	11110	
9	11001	
10	10111	
11	01011	
12	10110	
13	01001	
14	10010	
15	00001	
16	00010	
17	00100	
18	01000	
19	10000	
20	00101	
21	01010	
22	10100	
23	01101	
24	11010	
25	10001	
26	00111	
27	01110	
28	11100	
29	11101	
30	11111	
31	11011	
32	10011	

When using $p_1(x)$, we need seven distinct initial values to cover all possible patterns (See Table 3.2). Let cover set S_i denote the set of patterns that can be generated with initial value i. Let $\|S_i\|$ be the norm of S_i, or the number of elements in S_i or periodicity. Notice that $S_{21} = \{21\}$ is the only case that the periodicity is 1. When a state cover set has only one number, (periodicity 1), we call that an inertial state. Several properties of the cover sets are the following:

1. If $a \in S_i$ then $S_a = S_i$. Note that since the LFSR is a deterministic device and the next state q' of current state q is uniquely determined, therefore if $q \in S_i$ then $q' \in S_i$, when $i \neq 0$.

2. If $i \neq j$ then either $S_i \cap S_j = \varnothing$ or $S_i = S_j$. [proof:] $\exists q_i$ such that $q_i \in S_i$, $q_i \in S_j$ for $i \neq j$ then $\forall q \in S_i$, $q \in S_j$. Because the state transition is deterministic so the transitions in both S_i and S_j are in unison: n cycles after q_i should be in both sets or $q_i(n) = q \in S_i$ and $q_i(n) = q \in S_j$.

3. $\|S_i\| < 2^n - 1$ since the maximum number of states of n stages is $2^n - 1$. Therefore,

$$\bigcup_{i=1}^{2^{n-1}} S_i = \{1, 2, 3, \ldots, 2^n - 1\}$$

TABLE 3.2 THE COVER SETS GENERATED BY DIFFERENT INITIAL VALUE

Initial Value	Periodic Sequence Generated
00001 {1}	00001 00010 00100 01000 10000 11111 {1, 2, 4, 8, 16, 31}
3	3, 6, 12, 24, 15, 30
5	5, 10, 20, 23, 17, 29
7	7, 14, 28
9	9, 18, 27
19	19, 25, 13, 26, 11, 22
21	21

C. PSEUDORANDOM PATTERN GENERATOR

The most popular hardware pseudorandom pattern generator is the Linear Feedback Shift Register (LFSR). An LFSR composed of master-slave flip-flops and XOR gates. An LFSR implements a pseudorandom generator by feeding back a set of intermediate stages to the input. The general configuration is to add (modulo-2) all feedbacks to the input. The number of feedback terms are dictated by the number of terms in the characteristic polynomial p(x). For example, if $p(x) = x^5 + x^4 + x^3 + x + 1$, its LFSR can be implemented as Fig. 3.2.

Note that there is only one multiple input XOR gate in Fig. 3.2. A CMOS XOR gate with multiple inputs is more economical only in schematic not in implementation. Due to the regularity requirement in the VLSI, a designer will not use this scheme. If we were to layout an XOR gate of more than 3 inputs the layout would be much larger than the neighboring flip-flop. If we were to replace a multiple inputs XOR gate with multiple 2-input XOR gates, this feedback circuit for modulo-2 addition would again become larger than the neighboring flip-flop. Fig. 3.3 demonstrates such an arrangement; the routing wires to the XOR gates was dictated by the inherent nature of the tree structure similar to that of a parity tree. An n-input XOR gate can be implemented by (n – 1) 2-input XOR gates. For example, 4-input XOR can be expressed as: XOR(a, b, c, d) = XOR(XOR(a, b), XOR(c, d)). The alternative LFSR could be implemented as in Fig. 3.4. The beauty of this scheme is that although we use the same number of multiple 2-input XOR gates as that of Fig. 3.3 the area requirement is more economical: each XOR gates can be of the same size as that of a flip-flop. The biggest savings are the routing area

and the regularity of Fig. 3.4. The realization of Fig. 3.4 has less interstage delay than that of Fig. 3.3. Consequently, Fig. 3.4 could operate at a higher clock rate than Fig. 3.3.

With two basic leaf cells of a master-slave flip-flop and a 2-input XOR gate we can easily compose an LFSR to implement a given characteristic polynomial. Section 3.5 will discuss a generator that generates the LFSR layout for users.

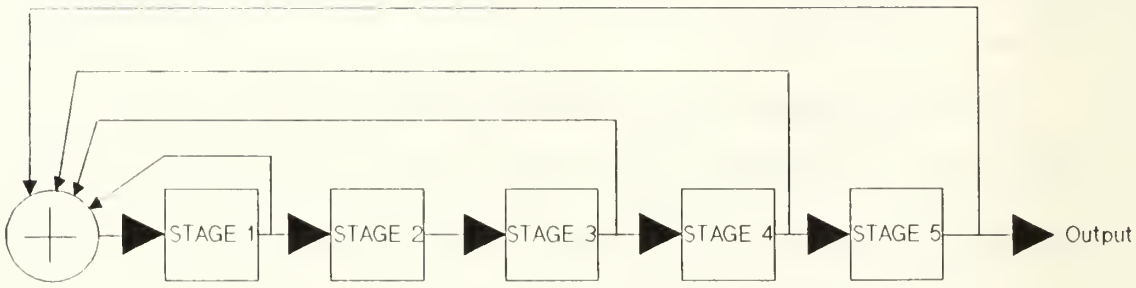

Figure 3.2 LFSR Implementing $x^5 + x^4 + x^3 + x + 1$

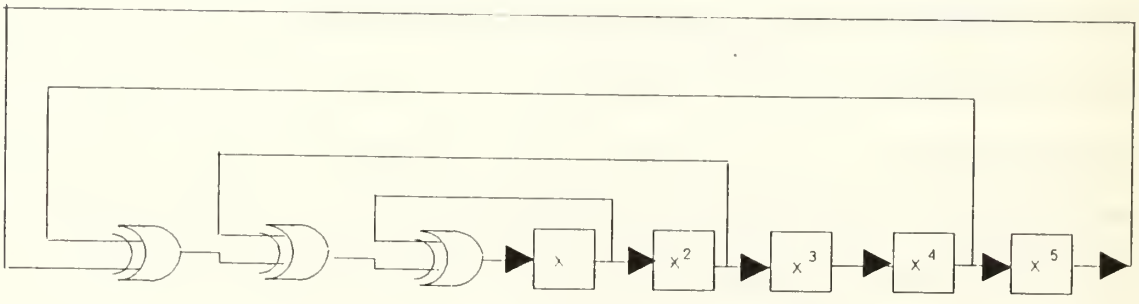

Figure 3.3 Different LFSR Implementing $x^5 + x^4 + x^3 + x + 1$

It is interesting to note the difference between Fig. 3.5 and Fig. 3.4 for a given characteristic polynomial: a CRC divider has one more XOR gate than an LFSR. From this observation, we can use a CRC divider to implement pseudorandom generator and the input XOR gate is used only for initialization. When the input stream of CRC divider is zero the input XOR gate is equivalent to a short circuit and therefore a CRC divider reduces to a LFSR pseudorandom pattern generator. On the other hand, for practical purposes, we need to provide a mechanism for initializing an LFSR so an input is required and we will implement an LFSR like a CRC divider. Therefore, in this thesis we treat the implementation of a CRC divider as a substitute for an LFSR pseudorandom generator.

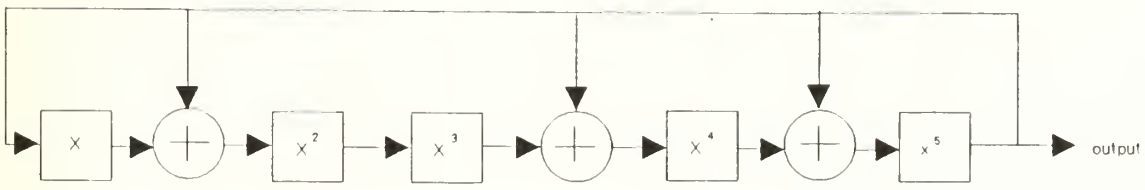

Figure 3.4 Alternate LFSR Implementation of $x^5 + x^4 + x^3 + x + 1$

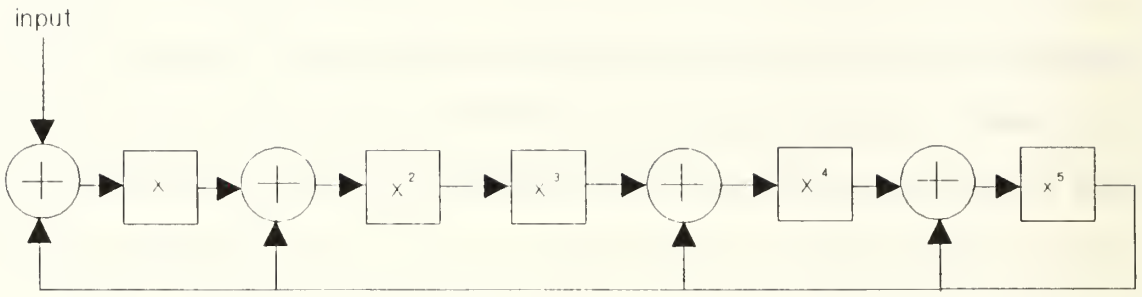

Figure 3.5 CRC Implementing $x^5 + x^4 + x^3 + x + 1$

D. NETLIST GENERATOR FOR CRC DIVIDERS

This section describes a netlist generator for CRC dividers. A "C" program named crcnet.c was written as part of the thesis research (see Appendix A). This program is used to parse the user input and produce the corresponding netlist px.net and its switch-level stimulation file px.l where px is the file name given by the user. Using the simulation file we can simulate the behavior of the CRC divider. A typical usage of the crcnet.c is shown in Appendix B for the generation of a CRC divider like the one in Fig. 3.5 by executing a unix command:

% crcnet 111011 10101 x54310

where 111011 is the representative of the polynomial $x^5 + x^4 + x^3 + x + 1$, 10101 is the initial value and x54310 is the filename of the two output files produced, x54310.net and x54310.l.

The network description file produced by crcnet is converted to an intermediate file, px.sim by using the NETLIST command. Another conversion occurs when producing a binary file suitable for use in RNL and is done by the PRESIM commands. The command file px.l should load one or more libraries of standard functions and should read in the binary description of the circuit created by PRESIM. The command file should also contain LISP commands and definitions required for the circuit functional simulation [Ref. 12]. This netlist simulator can provide the user a validation of certain polynomial's behavior. Besides, it can be used to check against the simulation based on the circuit extraction from the layout.

E. CRC LAYOUT GENERATOR

A netlist is used to describe a circuit and the simulation on a netlist is to verify the functional correctness of the circuit. To realize a circuit on a silicon chip we need to have the layout of the circuit. A CRC layout generator was written as a part of this research that can generate CMOS mask specifications for a given CRC divider polynomial.

This program, called crcmag.c, is used to get user's input and generate the corresponding layout file px.mag. The input is similar to that required by crcnet. Fig. 3.6 shows the CRC divider for $x^3 + x + 1$ generated by crcmag.c.

The area is 51,660 square units (630 x 82). The actual area of a layout depends not only on the order of the polynomial, but also on the number of terms. To calculate the area , the following formula can be used: $N \bullet L_1 + M \bullet L_2$ assuming both MSFF and XOR are of the same height. In the formula the N is the order of the polynomial, the M is the number of terms present , L_1 is the width of the MSFF (141), L_2 is the width of the XOR (100). Once the

layout is generated, extractor is used to convert the graphical layout into a circuit extraction px.ext file which contain information about the layout environment, geometry, and connectivity. This .ext file is converted to an intermediate file (the .sim file) by using the unix command EXT2SIM. Similar to the netlist generator approach, the unix command PRESIM is used to convert a .sim file to a binary file to be used in RNL. Table 3.3 shows an example of exhaustive test patterns generated by both crcnet and crcmag implementing $x^3 + x + 1$. Notice that both the netlist and the layout-extracted circuit produce identical patterns. In other words, the crcmag.c can produce a layout that realize the circuit behavior described on simulated earlier by crcnet.c.

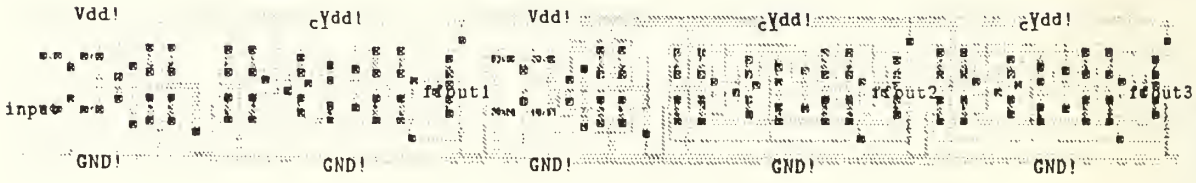

Figure 3.6 CRC Generate by MAGIC

TABLE 3.3 TEST PATTERN GENERATED BY NETLIST AND LAYOUT

cycle	Test Pattern generated by crcnet	Test Pattern generated by crcmag
1	101	101
2	001	001
3	010	010
4	100	100
5	011	011
6	110	110
7	111	111
8	101	101

IV. WEIGHTED PRPG

A useful extension of pseudo-random testing is weighted pseudo-random testing. In this approach the LFSRs are modified so they can supply unequal distribution of '1's and '0's per input bit of test patterns. The concept of weighted pseudorandom patterns (WPRP) was introduced in the early 1970's by [Ref. 7]. In this chapter we describe two methods of designing WPRP generators: netlist and circuit layout. Besides, we compute the output signal probabilities.

A. OVERVIEW OF WPRPG

To reduce the number of test patterns required for high fault coverage the technique named Weighted Pseudorandom Pattern Generator (WPRPG) has been used in built-in self test. WPRPG testing is a way to provide control to a random test set to achieve a maximum fault coverage in relatively small number of patterns [Ref. 13]. This control is achieved by adjusting the probability of the random values being applied at each input of the CUT. [Ref. 14] has indicated that the weighted random patterns can be generated internal to the device by additional circuitry at the output of an LFSR. In this method, the adaptive mechanism is provided by monitoring the logic level transition activity of the CUT [Ref. 13]. A weight is then assigned to a primary input of the CUT based on the amount of switching activity produced inside the logic as a result of exercising that particular primary input. In practice, the WPRPG contains 10 to 50 times less test patterns than a deterministic test, which is useful in improving the fault coverage of defects. Furthermore, when used

with a fast fault simulator, it has been shown that WPRPG is a highly efficient form of test generation [Ref. 13].

B. SIGNAL PROBABILITY

The pseudorandom patterns generated by the LFSR can be weighted to alter the frequency of 1's to a fraction other than 0.5. The pseudorandom patterns developed by the LFSR are weighted by logically ANDing and ORing the output lines of the LFSR to change the frequency of occurrence of one's and zero's. An AND gate increases the probability of occurrences of zeros. The occurrence of one's is increase by the addition of an OR gate. In achieving the occurrence of one's with probabilities other than 0.5, a network of AND and OR gates is employed at the outputs of the LFSR. The derivation of specific signal probabilities is achieved through two basic analytical probability formulas:

$$P(A \cap B) = P(A)P(B) ,$$

$$P(A \cup B) = P(A) + P(B) - P(A)P(B).$$

We have observed that it is not always efficient or even possible to reach exact signal probabilities using the above formulas. However, in weighted test pattern generation, iterative use of these formulas leads to accepted signal probabilities. As an example, if the outputs of two LFSR stages are fed into an AND gate, the output will have a signal probability approaching 0.25. The underlying assumption is that the signal probability is 0.5 for any stage in an LFSR. The circuit in Fig. 4.1, shows the basic concepts [Ref. 11]. The weight, or signal probability generated by WPRPG can be controlled by the contents of

the control register. The AND gate whose output is labelled 2 has inputs from two stages of the LFSR. If bit number 3 of control register contains a 1, AND gate 2 will feed a 1 to the OR gate 25% of the time since 0.5 x 0.5 = 0.25. If bit number 3 becomes 0, AND gate 2 will be disabled and does not give any effect to the signal probability of output. The output of three AND gates, which contribute 0.5, 0.25, or 0.125, respectively, are under control of the value of control register. The XOR is driven by bit number 1 of the control register and serves as an inverter for the output of the OR gate.

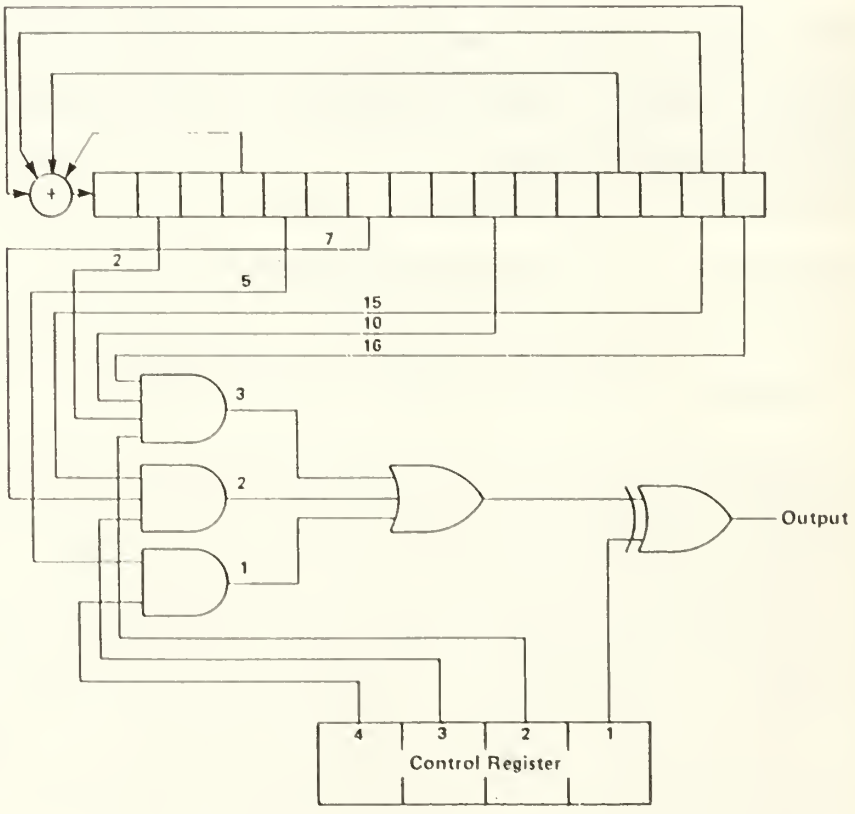

Figure 4.1 Programmable Weighted Random Pattern Generator

C. TWO EXPERIMENTS

1. WPRPG Netlist Simulation

In order to generate WPRPG using Netlist, two programs are written: GLFSR and GWPRPG. The GLFSR generates an LFSR for the polynomial desired. The GWPRPG connects the additional circuit needed as mentioned in previous section into one netlist module called WPRPG.net. A set of experiments are repeated by using WPRPG.net.

2. Circuit Layout

The configuration of WPRPG shown in Fig. 4.1 consists of LFSR, 3 AND gates, 1 OR gate, XOR gate and 4 bits control register. This circuit layout takes a silicon area of 462 x 1424 square units in scalable 3μ CMOS. Similar to the unbiased LFSR discussed in Chapter III, this layout is extracted and simulated. The simulation results are validated by the netlist counterparts. For example, an LFSR of $x^{16} + x^5 + x^3 + x^2 + 1$ has been clocked 700 cycles separately for initial values of 1011000000000000 and 1000000000000000. Table 4.1 shows probabilities using different control register values when use 1011000000000000 as initial value of the LFSR. For example, loading 1000 into the control register, we obtain the signal probability of 0.501429 which is greater than the theoretical 0.5. When we use 0100 as the control register value the signal probability is less than theoretical 0.25. The discrepancies here are due to the number of cycles we simulated. In order to obtain the accurate signal probability value, we have to increase the cycles of simulation to $2^{16} - 1$.

Table 4.2 shows the probabilities with initial LFSR value of 1000000000000000. Comparing Table 4.1 and Table 4.2, we notice that different

initial values of LFSR may result in different signal probabilities. However, they both close to expected values. Therefore initial value of LFSR did not influence the probability in WPRPG. In other words, the output signal probabilities are controlled by the control register contents only. Finally, all of the experiment done above have been repeated using circuits created by the netlist generator. From the netlist generator, Table 4.3 shows the netlist simulation results that validates the WPRP circuit generated by the layout generator (Table 4.1). Similarly, Table 4.4 validates Table 4.2. As a conclusion, weighted pseudorandom pattern generation may generate less patterns than exhaustive test pattern generation at a price of extra silicon area for those AND, OR gates and control register. The usefulness of this biased test pattern generation depends on the tradeoffs between the pattern generation time and silicon area.

TABLE 4.1 THE PROBABILITIES USING 1011000000000000 AS INITIAL VALUES OF LFSR (LAYOUT GENERATOR)

Control Register	Simulated Probability	Expected Probability
1000	0.501429	0.5
0100	0.248571	0.25
0010	0.112857	0.125
0101	0.751429	0.75
0011	0.887143	0.875

TABLE 4.2 THE PROBABILITIES USING 1000000000000000 AS INITIAL VALUES OF LFSR (LAYOUT GENERATOR)

Control Register	Simulated Probability	Expected Probability
1000	0.478571	0.5
0100	0.222857	0.25
0010	0.111429	0.125
0101	0.777143	0.75
0011	0.888571	0.875

TABLE 4.3 THE PROBABILITIES USING 1011000000000000 AS INITIAL VALUES OF LFSR (NETLIST GENERATOR)

Control Register	Simulated Probability	Expected Probability
1000	0.501429	0.5
0100	0.248571	0.25
0010	0.112857	0.125
0101	0.751429	0.75
0011	0.887143	0.875

TABLE 4.4 THE PROBABILITIES USING 1000000000000000 AS INITIAL VALUES OF LFSR (NETLIST GENERATOR)

Control Register	Simulated Probability	Expected Probability
1000	0.478571	0.5
0100	0.222857	0.25
0010	0.111429	0.125
0101	0.777143	0.75
0011	0.888571	0.875

V CONCLUSIONS

This thesis has introduced an economical way of implementing the test pattern generation for built-in test: the internal form of LFSR. A layout generator as well as a netlist generator are written and validated. Similar generators have been written for weighted LFSR as well. The layout generator allows the user to produce mask specifications for test pattern generators that the circuit under test requires. In addition to validating the layout, one major contribution of the netlist generator is to explore various characteristics of a given generator polynomial. The generators written for this thesis have been used in another thesis work by Barus [Ref. 5]. Barus' work is developed concurrently with this thesis so there is a strong correlation between this thesis and Barus' work. With a basic understanding of these two theses work, one may gain an overall picture of the built-in test.

The primitive polynomials have been studied by many researches, although nonprimitive ones are orphans in the research circle. In this thesis, an investigation into properties of nonprimitive polynomials has been performed using netlist generator described in Chapter III. The polynomials of degrees 3, 4 and 5 are summarized in Appendix C. There are some interesting properties or observations:

1. the existence of inertial state that stay unchanged;
2. the nonprimitive cycles are either disjoint or coincident;
3. each primitive or nonprimitive cycle follows the sequence that every successor of a state i is $2i$ until the successor is greater than $2^n - 1$ and

then drops to a new state; the new state is dictated by the polynomial or feedback circuit.

On the other hand, there are some conjectures that we leave here as open problems for further research:

1. for a given nonprimitive polynomial p(x) how to find an initial state that gives the maximum nonprimitive cycle,
2. for a given p(x), how to find an inertial state if any,
3. what is the minimum change to the p(x) above such that we can modify it into a new primitive polynomial p(x),
4. given a cycle or a finite set of test patterns, how to find a nonprimitive polynomials that generates the cycle.

The last problem (4) above is important since we may want to have maximum fault coverage using only this nonexhaustive cycle when the circuit behavior does not require exhaustive patterns. Our current conjecture is that the solution might exist but may not be unique.

APPENDIX A: PROGRAM LISTINGS

A. CRCNET

```
/*
-------------------------------------------------
PROGRAM NAME    : CRCNET
APPLICATION     : To generate and simulate CRC for
                  any polynomial
-------------------------------------------------
*/

#include <stdio.h>
#define fname_len 20
#define len 200
#define MAXDEG 100
#define newline fprintf(fid, "\n")

char initial[MAXDEG*2];

main(argc, argv)
int argc;
char **argv;

{
FILE *fid;
char filebase[fname_len], memory[40];
char fb[fname_len];
char fc[fname_len];
char fs[fname_len];
char inp;
int i, j, N=0, M, I=0, T;
int xor[MAXDEG];
int max[16];
int sw1 = 0;
int c0 = 1, c1 = 1;
int c2 = 1, c3 = 1;
int c4 = 1, c5 = 1;
int c6 = 1, c7 = 2;
int c8 = 1, c9 = 1;
int k  = 1, m  = 1;


strcpy(filebase, argv[3]);
strcpy(fb,filebase);
strcpy(fs,filebase);
strcpy(fc,filebase);
```

```
if(argc !=4)
  {
   printf("Usage: crcnet polynomial initial filename\n");
   printf("Example : crcnet 111011  10101 file\n");
   printf("          for x^5+x^4+x^3+x+1 with 10101\n");
   exit(1);
  }

N = strlen(argv[1]) - 1;
for(i=0; i<=N; i++) max[N-i] = argv[1][i] - 48; /* ascii 0 = 48 */

M = N;

I = strlen(argv[1]);   /* I is the degree minus 1 */

strcpy(initial,"     ");
for(i=0; i< I; i++)
 {
  if(argv[2][i]==49)  /* ascii '1' has value 49 */
    {
     sprintf(memory,"h %d h %d ", 2*i, 2*i+1);
     strcat(initial, memory);

    }
  else {
      if(argv[2][i]==48)
       {  sprintf(memory,"l %d l %d ", 2*i, 2*i+1);
          strcat(initial, memory);

}}
}  /* end for */

/*
--------------------------------
CREATE NETWORK DESCRIPTION FILE.
--------------------------------
*/

strcat(filebase,".net");
fid= fopen(filebase,"w");

fprintf(fid, "(load \"lib.net\")\n");
fprintf(fid, "(load \"xor2.net\")\n");
newline;
fprintf(fid, "("); newline;
```

```
fprintf(fid, "node cl ");
newline; newline;
for(i=1; i<=N; i++) {
                     fprintf(fid, " ffout%d ",i);
                     newline;
                    }
newline;
for(i=1; i<=N; i++) {
                     fprintf(fid, " ffin%d ",i);
                     newline;
                    }
newline;
                     fprintf(fid, " input");
newline;
newline;

max[N] = max[0] = 1;


/* to verify the polynomial */
   printf(" \n\n\tX^%d + ",N);
   for (i = N-1; i>0; i--)
       {
        if (max[i] == 1)
         printf("X^%d + ",i);
       }
   printf("1\n");

   for (i=N-1; i>0; i--)  if(max[i] == 1)   k++;

for(j=1; j<=k; j++)
   {
    fprintf(fid, "xorout%d xorin%d1 xorin%d2 ",j,j,j);
    newline;
   }
newline;
fprintf(fid, ") "); newline; newline;


for(i=1; i<=N; i++) fprintf(fid, "(msff ffout%d ffin%d cl)\n",i, i);
newline;

    for(i=1; i<=k; i++)
    {
    fprintf(fid, "(xor2 xorout%d xorin%d1 xorin%d2 )\n",i,i,i);
```

```
    }
    newline;


         for (i=N-1; i>0; i--)
         {
         if (max[i] == 1)
           {
          fprintf(fid, "(connect ffin%d xorout%d)",N,c2); newline;
          c2++;
          fprintf(fid, "(connect xorin%d1 ffout%d)",c8,N-1); newline;
          fprintf(fid, "(connect xorin%d2 ffout%d)",c8,M); newline;
          newline;
          newline;
          c8++;
          N--;
           }
         else if (max[i] == 0)
           {
                fprintf(fid, "(connect ffin%d ffout%d)",N,N-1); newline;
                newline;
                N--;
           }
         }
        fprintf(fid, "(connect ffin%d xorout%d)",N,c2); newline;
        fprintf(fid, "(connect xorin%d1 ffout%d)",c8,M); newline;
    fprintf(fid, "(connect xorin%d2 input)",c8); newline;
fclose(fid);

/*
-------------------------------------------------------------
CREATE A FLATTENED NETLIST REPRESENTATION AND A BINARY CIRCUIT
REPRESENTATION.
-------------------------------------------------------------
*/

sprintf(memory,"netlist %s.net  %s.sim -tcmos-pw",fb,fb);
system(memory);
sprintf(memory,"presim %s.sim %s",fb,fb);
system(memory);

/*
-------------------------------------------------------
CONVERT A SIMPLE TIMING FILE IN A RNL COMPATIBLE FORMAT.
-------------------------------------------------------
```

```
*/

strcat(fc,".stim");
fid=fopen(fc,"w");

fprintf(fid, "time_range 0 100"); newline;

fprintf(fid, "cl 2 1 0 h 1"); newline;

fprintf(fid, "input 0 %s",initial); newline;

fprintf(fid, "report 1 0"); newline;
fclose(fid);
sprintf(memory, "gen_time %s.stim %s.time",fs,fs);
system(memory);

/*
-----------------------------------------------------------------
CREATE A FILE THAT CONTAINS A SEQUENCE OF RNL COMMAND AND SIMULATE.
-----------------------------------------------------------------
*/

strcat(fb,".l");
fid=fopen(fb,"w");

fprintf(fid, "(load \"uwstd.l\")"); newline;
fprintf(fid, "(load \"uwsim.l\")"); newline;
fprintf(fid, "(log-file \"%s.rlog\")",fs); newline;
fprintf(fid, "(read-network \"%s\")",fs); newline;
fprintf(fid, "(setq incr 100)"); newline;
fprintf(fid, "(sim-init)"); newline;
fprintf(fid, "(sim-init)"); newline;

fprintf(fid, "(defvec '(bit rem ");
for(i=M; i>0; i--)
  {
   fprintf(fid, "ffout%d ",i);
  }
fprintf(fid, "))");
newline;
fprintf(fid, "(def-report '(\" \" cl ");
fprintf(fid, "input ");
fprintf(fid, "(vec rem)))"); newline;
fprintf(fid, "(setq lanalyze t)"); newline;
fprintf(fid, "(wr-format)"); newline;
```

```
fprintf(fid, "(load \"%s.time\")",fs); newline;
fprintf(fid, "exit"); newline;
fclose(fid);
sprintf(memory, "rnl %s.l",fs);
system(memory);
sprintf(memory, "print %s.rlog",fs);
system(memory);


}
```

B. CRCMAG

```
/*
----------------------------------------
program name     : crcmag; layout generator
input            : LFSR polynomial
output           : layout
application      : to generate crc layout
----------------------------------------
*/

#include <stdio.h>
#include "cfl.h"

main()
{
      /*define the variables */
SYMBOL *s0, *s1, *s2, *s3 ;
int    term_present[16], order=0, i;

do
{
  printf("\n\tinput order: ");
  scanf("%d",&order);
}
while (order == 0);
term_present[order] = 1;
term_present[0]=1;
for (i=order-1; i>0; i--)
{
printf("n\tx^%d term - type '1' if present and '0' if not: ",i);
scanf("%d",&term_present[i]);
if (term_present[i]!=1)
term_present[i]=0;
}

printf("\n\n\tX^%d + ",order);
for (i=order-1; i>0; i--)
{
  if (term_present[i]==1)
      printf("X^%d + ",i);
}
printf("1\n");

cflsetc("format","magic");
cflstart("scmos");
```

```
cflsetv("grain", 1);

       /* pieces of poly and poly with sizes */
s0=box("polysilicon",2,6);
       /* leaf cells  (./mag location implied) */
s1=gs("msff3");
s3=s2=gs("xor3");
s3=bb(s2,s1);

for (i=1; i<order; i++)
{
  if (term_present[i]==1)
    {
     s3=bb(s3,s2);
     printf("\n\tTEST",i);
    }
  s3=bb(s3,s1);
}
s3=bbdx(s3,s0,-1);

       /* name the crcmag.mag */

ps("crcmag",s3);

cflstop();
}
```

C. GLFSR

```
/*
-------------------------------------------------
program name    : glfsr
application     : to generate and simulate lfsr for
                  any polynomial
-------------------------------------------------
*/

#include <stdio.h>
#define fname_len 20
#define len 200
#define MAXDEG 100
#define newline fprintf(fid, "\n")

char initial[MAXDEG*2];

main(argc, argv)
int argc;
char **argv;

{
FILE *fid;
char filebase[fname_len], memory[40];
char fb[fname_len];
char fc[fname_len];
char fs[fname_len];
int i, j, N=0, M, I=0, T;
int xor[MAXDEG];
int max[16];
int c0 = 1, c1 = 1;
int c2 = 1, c3 = 1;
int c4 = 1, c5 = 1;
int c6 = 1, c7 = 2;
int c8 = 1, c9 = 1;
int k  = 0, m  = 0;
int p  = 1, q  = 0;

strcpy(filebase, argv[3]);
strcpy(fb,filebase);
strcpy(fs,filebase);
strcpy(fc,filebase);

if(argc !=4)
  {
```

```
   printf("Usage: glfsr polynomial initial filename\n");
   printf("Example : glfsr 111011 10101 file\n");
   printf("          for x^5+x^4+x^3+x+1 with 10101\n");
   exit(1);
  }
N = strlen(argv[1]) - 1;
for(i=0; i<=N; i++) max[N-i] = argv[1][i] - 48; /* ascii 0 = 48 */
M = N;

I = strlen(argv[1]); /* I is the degree minus 1 */

strcpy(initial,"       ");
for(i=0; i<I; i++)
 {
  if(argv[2][i]==49) /* ascii '1' has value 49 */
    {
     sprintf(memory,"h %d h %d ", 2*i, 2*i+1);
     strcat(initial, memory);
     q  = q + 2*i+1;
printf("%s\n", initial);
    }
 else {
       if(argv[2][i]==48)
    {
     sprintf(memory,"l %d l %d ", 2*i, 2*i+1);
     strcat(initial, memory);
     q = q + 2*i+1;
printf("%s\n", initial);
    }}
 } /* end for */

/*
--------------------------------
CREATE NETWORK DESCRIPTION FILE.
--------------------------------
*/

strcat(filebase,".net");
fid= fopen(filebase,"w");

fprintf(fid, "(load \"lib.net\")\n");
fprintf(fid, "(load \"xor2.net\")\n");
newline;
fprintf(fid, "("); newline;
fprintf(fid, "node cl ");
```

```
newline; newline;
for(i=1; i<=N; i++) {
                     fprintf(fid, " ffout%d ",i);
                     newline;
                    }
newline;
for(i=1; i<=N; i++) {
                     fprintf(fid, " ffin%d ",i);
                     newline;
                    }
newline;
                     fprintf(fid, " input");
newline;
newline;

max[N] = 1;
max[0] = 1;


/* to verify the polynomial */
   printf(" \n\n\tX^%d + ",N);
   for (i = N-1; i>0; i--)
    {
     if (max[i] == 1)
     printf("X^%d + ",i);
    }
     printf("1\n");

     for (i=N-1; i>0; i--)
      {
       if(max[i] == 1)
       {
        k++;
       }
       else if (max[i] == 0)
        {
         m++;
        }
      }

for(j=1; j<=k; j++)
   {
    fprintf(fid, "xorout%d xorin%d1 xorin%d2 ",j,j,j);
    newline;
   }
```

```
newline;
fprintf(fid, ") "); newline; newline;


for(i=1; i<=N; i++) fprintf(fid, "(msff ffout%d ffin%d cl)\n",i, i);
newline;

    for(i=1; i<=k; i++)
    {
    fprintf(fid, "(xor2 xorout%d xorin%d1 xorin%d2 )\n",i,i,i);
    }
    newline;

/*
INSERT NEW PROGRAM
*/

for(i=1; i<=k-1; i++)
 {
  fprintf(fid, "(connect xorout%d xorin%d2)\n",i,i+1); newline;
  newline;
  m=i+1;
 }
       if(k==1)
                {
                m=1;
                fprintf(fid, "(connect xorout%d ffin%d)\n",m,p); newline;
                }
       else if(k>1)
                {
                fprintf(fid, "(connect xorout%d ffin%d)\n",m,p); newline;
                }

         for (i=1; i<=N-1; i++)
          {
          if (max[i] == 1)
           {
                fprintf(fid, "(connect ffout%d xorin%d1)",p,k); newline;
                fprintf(fid, "(connect ffout%d ffin%d)",p,p+1); newline;
                newline;
                p++;
                k--;
           }
          else if (max[i] == 0)
           {
```

```
                fprintf(fid, "(connect ffout%d ffin%d)",p,p+1); newline;
                newline;
                p++;
            }
           }
        fprintf(fid, "(connect ffout%d xorin12)",N); newline;
fclose(fid);

/*
-------------------------------------------------------------
CREATE A FLATTENED NETLIST REPRESENTATION AND A BINARY CIRCUIT
REPRESENTATION.
-------------------------------------------------------------
*/

sprintf(memory,"netlist %s.net  %s.sim -tcmos-pw",fb,fb);
system(memory);
sprintf(memory,"presim %s.sim %s",fb,fb);
system(memory);

/*
-----------------------------------------------------
CONVERT A SIMPLE TIMING FILE IN A RNL COMPATIBLE FORMAT.
-----------------------------------------------------
*/

strcat(fc,".stim");
fid=fopen(fc,"w");

fprintf(fid, "time_range 0 50"); newline;

fprintf(fid, "cl 2 1 0 h 1"); newline;

fprintf(fid, "ffout%d 1 %s x %d",N,initial,q/2); newline;

fprintf(fid, "report 1 0"); newline;
fclose(fid);
sprintf(memory, "gen_time %s.stim %s.time",fs,fs);
system(memory);

/*
-------------------------------------------------------------
CREATE A FILE THAT CONTAINS A SEQUENCE OF RNL COMMAND AND SIMULATE.
-------------------------------------------------------------
*/
```

```
strcat(fb,".l");
fid=fopen(fb,"w");

fprintf(fid, "(load \"uwstd.l\")"); newline;
fprintf(fid, "(load \"uwsim.l\")"); newline;
fprintf(fid, "(log-file \"%s.rlog\")",fs); newline;
fprintf(fid, "(read-network \"%s\")",fs); newline;
fprintf(fid, "(setq incr 100)"); newline;
fprintf(fid, "(sim-init)"); newline;
fprintf(fid, "(sim-init)"); newline;

fprintf(fid, "(defvec '(bit rem ");
for(i=1; i<=M; i++)
  {
   fprintf(fid, "ffout%d ",i);
  }
fprintf(fid, "))");
newline;
fprintf(fid, "(def-report '(\" \" cl ");

fprintf(fid, "ffout%d",N);

fprintf(fid, "(vec rem)))"); newline;
fprintf(fid, "(setq lanalyze t)"); newline;
fprintf(fid, "(wr-format)"); newline;
fprintf(fid, "(load \"%s.time\")",fs); newline;
fprintf(fid, "exit"); newline;
fclose(fid);
sprintf(memory, "rnl %s.l",fs);
system(memory);
sprintf(memory, "print %s.rlog",fs);
system(memory);
}
```

D. GWPRPG

```
/*
-------------------------------------------------------------
program name : gwprpg
application  : 1. To connect LFSR and auxiliary circuit
                  for generating different signal probability.
                  Note that the LFSR should be of degree 16 or
                  higher.
               2. Having created the weighted LFSR, netlist,
                  this program starts the execution of the
                  netlist simulation.
-------------------------------------------------------------
*/

#include <stdio.h>
#define fname_len 20
#define len 200
#define MAXDEG 300
#define newline fprintf(fid, "\n")

main(argc, argv)
int argc;
char **argv;

{
FILE *fid;
char filebase[fname_len], memory[40];
char fb[fname_len];
char fc[fname_len];
char fs[fname_len];
char fa[fname_len];
char fm[fname_len];
char fl[fname_len];

char *ch;
int i, j, d, g;
int k=1, N=0, M, T,l=0;
int xor[MAXDEG];
int max[16];

printf("\n\tEnter file name for lfsr           : ");
scanf("%s",fl);

printf("\n\tEnter file name for nand           : ");
scanf("%s",fa);
```

```
printf("\n\tEnter file name for prpg            : ");
scanf("%s",filebase);
strcpy(fb,filebase);
strcpy(fs,filebase);
strcpy(fc,filebase);


/* create file .stim and generate file .time */
/* ----------------------------------------- */

strcat(fc,".stim");
fid=fopen(fc,"w");

fprintf(fid, "time_range 0 1000"); newline;

fprintf(fid, "cl 2 h 0 l 1"); newline;
fprintf(fid, "input 0 h 0 h 1 l 2 l 3 h 4 h 5 l 6 ");
newline;
printf("\n\tEnter number of bits number 4       : ");
scanf("%d",&i);
if(i==1)
    {
    fprintf(fid, "i 2 h 0 h 1"); newline;
    }
     else
     {
     fprintf(fid, "i 2 l 0 l 1"); newline;
     }

printf("\n\tEnter number of bits number 3       : ");
scanf("%d",&g);
if(g==1)
    {
    fprintf(fid, "g 2 h 0 h 1"); newline;
    }
     else
     {
     fprintf(fid, "g 2 l 0 l 1"); newline;
     }

printf("\n\tEnter number of bits number 2       : ");
scanf("%d",&d);
if(d==1)
    {
```

```
    fprintf(fid, "d 2 h 0 h 1"); newline;
    }
     else
     {
     fprintf(fid, "d 2 1 0 1 1"); newline;
     }

printf("\n\tEnter number of bits number 1       : ");
scanf("%d",&j);
if(j==1)
    {
    fprintf(fid, "j 2 h 0 h 1"); newline;
    }
     else
     {
     fprintf(fid, "j 2 1 0 1 1"); newline;
     }

     fprintf(fid, "report 1 0"); newline;
fclose(fid);
sprintf(memory, "gen_time %s.stim %s.time",fs,fs);
system(memory);

/* create file .net, netlist and presim */
/* ------------------------------------ */

strcat(filebase,".net");
fid= fopen(filebase,"w");

fprintf(fid, "(load \"lib.net\")\n");
fprintf(fid, "(load \"%s.net\")\n",f1);
fprintf(fid, "(load \"%s.net\")\n",fa);
fprintf(fid, "(load \"xor2.net\")\n");
newline;
fprintf(fid, "(node i j d g)"); newline;

      fprintf(fid, "(connect ffout11 nandin411)"); newline;
      fprintf(fid, "(connect ffout12 nandin412)"); newline;
      fprintf(fid, "(connect ffout13 nandin421)"); newline;
      fprintf(fid, "(connect nandin422 d)"); newline;

      fprintf(fid, "(connect ffout14 nandin311)"); newline;
      fprintf(fid, "(connect ffout15 nandin312)"); newline;
      fprintf(fid, "(connect nandin322 g)"); newline;
```

```
      fprintf(fid, "(connect ffout16 nandin211)"); newline;
      fprintf(fid, "(connect nandin212 i)"); newline;
      fprintf(fid, "(connect xorin212 j)"); newline;

fclose(fid);
sprintf(memory,"netlist %s.net  %s.sim -tcmos-pw",fb,fb);
system(memory);
sprintf(memory,"presim %s.sim %s",fb,fb);
system(memory);

/* create file .l and simulate */
/* --------------------------- */

strcat(fb,".l");
fid=fopen(fb,"w");

fprintf(fid, "(load \"uwstd.l\")"); newline;
fprintf(fid, "(load \"uwsim.l\")"); newline;
fprintf(fid, "(log-file \"%s.rlog\")",fs); newline;
fprintf(fid, "(read-network \"%s\")",fs); newline;
fprintf(fid, "(setq incr 100)"); newline;
fprintf(fid, "(sim-init)"); newline;
fprintf(fid, "(sim-init)"); newline;

fprintf(fid, "(defvec '(bit rem ");
for(i=1; i<=N; i++)
  {
   fprintf(fid, "ffo%d ",i);
  }
   fprintf(fid, "))");
newline;

fprintf(fid, "(def-report '(\" \"  ");
fprintf(fid, "i g d j xorout21");
fprintf(fid, "(vec rem)))"); newline;
fprintf(fid, "(setq lanalyze t)"); newline;
fprintf(fid, "(wr-format)"); newline;
fprintf(fid, "(load \"%s.time\")",fs); newline;
fprintf(fid, "exit"); newline;
fclose(fid);
sprintf(memory, "rnl %s.l",fs);
system(memory);

/*
sprintf(memory, "print %s.rlog",fs);
```

```
system(memory);
*/

/* COUNT THE PROBABILITY */

sprintf(memory, "count %s.rlog",fs);
system(memory);
}
```

APPENDIX B:

The listing below shows one sample execution of netlist generator 'crcnet' on polynomial x^5 + x^4 + x^3 + x + 1 (111011), with initial value 10101 and filename x54310.

```
% crcnet 111011 10101 x54310

X^5 + X^4 + X^3 + X^1 + 1

Version 4.2
presim: can not open config file cmos-pw.typ.config,
   alternatively using /tools/nwlis/lib/technology/cmos-pw.typ.config
116 nodes; transistors: enh=80 intrinsic=0 p-chan=80 dep=0 low-power=0
Pullup=0 resistor=0
Total transistors eliminated = 128
Version 4.2
Loading uwsim.l
Done loading uwsim.l
; 76 nodes, transistors: enh=16 intrinsic=0 p-chan=16 dep=0 low-power=0
Pullup=0 resistor=0

; Report format of logic analyzer style output
 time  cl  input  rem

10   0 1 00000
20   1 1 00000
30   0 0 00001
40   1 0 00001
50   0 1 00010
60   1 1 00010
70   0 0 00101
80   1 0 00101
90   0 1 01010
100   1 1 01010
110   0 0 10101
120   1 0 10101
130   0 0 10001
140   1 0 10001
150   0 0 11001
160   1 0 11001
170   0 0 01001
180   1 0 01001
190   0 0 10010
200   1 0 10010
210   0 0 11111
```

```
220  1 0 11111
230  0 0 00101
240  1 0 00101
250  0 0 01010
260  1 0 01010
270  0 0 10100
280  1 0 10100
290  0 0 10011
300  1 0 10011
310  0 0 11101
320  1 0 11101
330  0 0 00001
340  1 0 00001
350  0 0 00010
360  1 0 00010
370  0 0 00100
380  1 0 00100
390  0 0 01000
400  1 0 01000
410  0 0 10000
420  1 0 10000
430  0 0 11011
440  1 0 11011
450  0 0 01101
460  1 0 01101
470  0 0 11010
480  1 0 11010
490  0 0 01111
500  1 0 01111
510  0 0 11110
520  1 0 11110
530  0 0 00111
540  1 0 00111
550  0 0 01110
560  1 0 01110
570  0 0 11100
580  1 0 11100
590  0 0 00011
600  1 0 00011
610  0 0 00110
620  1 0 00110
630  0 0 01100
640  1 0 01100
650  0 0 11000
660  1 0 11000
```

```
670   0 0 01011
680   1 0 01011
690   0 0 10110
700   1 0 10110
710   0 0 10111
720   1 0 10111
730   0 0 10101
740   1 0 10101
750   0 0 10001
760   1 0 10001
770   0 0 11001
780   1 0 11001
790   0 0 01001
800   1 0 01001
810   0 0 10010
820   1 0 10010
830   0 0 11111
840   1 0 11111
850   0 0 00101
860   1 0 00101
870   0 0 01010
880   1 0 01010
890   0 0 10100
900   1 0 10100
910   0 0 10011
920   1 0 10011
930   0 0 11101
940   1 0 11101
950   0 0 00001
960   1 0 00001
970   0 0 00010
980   1 0 00010
990   0 0 00100
1000   1 0 00100
```

APPENDIX C: COMPARISON NONPRIMITIVE AND PRIMITIVE SEQUENCE

Nonprimitive (3, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4
3	3, 6, 5
7	7
Nonprimitive (3, 2, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 7
3	3, 6
5	5
Primitive (3, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 3, 6, 7, 5
Primitive (3, 2, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 5, 7, 3, 6

Nonprimitive (4, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8
3	3, 6, 12, 9
5	5, 10
7	7, 14, 13, 11
15	15
Nonprimitive (4, 2, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 5, 10
3	3, 6, 12, 13, 15, 11
7	7, 14, 9
Nonprimitive (4, 2, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 7, 14, 11
3	3, 6, 12, 15, 9, 5, 10
13	13
Nonprimitive (4, 3, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 11, 13
3	3, 6, 12
5	5, 10, 15
7	7, 14
9	9

Nonprimitive (4, 3, 2, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 13, 7, 14
3	3, 6, 12, 5, 10, 9, 15
11	11
Nonprimitive (4, 3, 2, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 15
3	3, 6, 12, 7, 14
5	5, 10, 11, 9, 13
Primitive (4, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9
Primitive (4, 3, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 9, 11, 15, 7, 14, 5, 10, 13, 3, 6, 12

Nonprimitive (5, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16
3	3, 6, 12, 24, 17
5	5, 10, 20, 9, 18
7	7, 14, 28, 25, 19
11	11, 22, 13, 26, 21
15	15, 30, 29, 27, 23
31	31
Nonprimitive (5, 4, 0)	
Initial Value	Periodic Sequence
1	1,2,4,8,16,17,19,23,31,15,30,13,26,5,10,20,25,3,6,,12,24
7	7, 14, 28, 9, 18, 21, 27
11	11, 22, 29
Nonprimitive (5, 1, 0)	
Initial Value	Periodic Sequence
1	1,2,4,8,16,3,6,12,24,19,5,10,20,11,22,15,30,31,29,25,17
9	9, 18, 7, 14, 28, 27, 21
13	13, 26, 23

Nonprimitive (5, 4, 3, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16, 25, 11, 22, 21, 19, 31, 7, 14, 28
3	3, 6, 12, 24, 9, 18, 29
5	5, 10, 20, 17, 27, 15, 30
13	13, 26
23	23
Nonprimitive (5, 3, 2, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16, 13, 26, 25, 31, 19, 11, 22
3	3, 6, 12, 24, 29, 23
5	5, 10, 20
7	7, 14, 28, 21
9	9, 18
15	15, 30, 17
27	27
Nonprimitive (5, 2, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16, 7, 14, 28, 31, 25, 21, 13, 26, 19
3	3, 6, 12, 24, 23, 9, 18
5	5, 10, 20, 15, 30, 27, 17
11	11, 22
29	29

Nonprimitive (5, 4, 2, 0)	
Initial Value	Periodic Sequence
1	1,2,4,8,16,21,31,11,22,25,7,14,28,13,26
3	3,6,12,24,5,10,20,29,15,30,9,18,17,23,27
19	19
Nonprimitive (5, 4, 3, 2, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16, 31
3	3, 6, 12, 24, 15, 30
5	5, 10, 20, 23, 17, 29
7	7, 14, 28,
9	9, 18, 27
11	11, 22, 19, 25, 13, 26
21	21

Nonprimitive (5, 3, 1, 0)	
Initial Value	Periodic Sequence
1	1,2,4,8,16,11,22,7,14,28,19,13,26,31,21
3	3,6,12,24,27,29,17,9,18,15,30,23,5,10,20
25	25
Nonprimitive (5, 4, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16, 19, 21, 25
3	3, 6, 12, 24
5	5, 10, 20, 27
7	7, 14, 28, 11, 22, 31, 13, 26
9	9, 18, 23, 29
15	15, 30
17	17
Nonprimitive (5, 4, 2, 0)	
Initial Value	Periodic Sequence
1	1,2,4,8,16,21,31,11,22,25,7,14,28,13,26
3	3,6,12,24,5,10,20,29,15,30,9,18,17,23,27
19	19
Nonprimitive (5, 4, 3, 2, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16, 31
3	3, 6, 12, 24, 15, 30
5	5, 10, 20, 23, 17, 29
7	7, 14, 28,
9	9, 18, 27
11	11, 22, 19, 25, 13, 26
21	21

Primitive (5, 3, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16, 9, 18, 13, 26, 29, 19, 15, 30, 21, 3, 6, 12, 24, 25, 27, 31, 23, 7, 14, 28, 17, 11, 22, 5, 10, 20

Primitive (5, 2, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16, 5, 10, 20, 13, 26, 17, 7, 14, 28, 29, 31, 27, 19, 3, 6, 12, 24, 21, 15, 30, 25, 23, 11, 22, 9, 18

Primitive (5, 4, 3, 2, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16, 29, 7, 14, 28, 5, 10, 20,21, 23, 19, 27, 11, 22, 17, 31, 3, 6, 12, 24, 13, 26, 9, 18, 25, 15, 30

Primitive (5, 3, 2, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16, 15, 30, 19, 9, 18, 11, 22, 3, 6, 12, 24, 31, 17, 13, 26, 27, 25, 29, 21, 5, 10, 20, 7, 14, 28, 23

Primitive (5, 4, 2, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16, 23, 25, 5, 10, 20, 31, 9, 18, 19, 17, 21, 29, 13, 26, 3, 6, 12, 24, 7, 14, 28, 15, 30, 11, 22, 27,

Primitive (5, 4, 3, 1, 0)	
Initial Value	Periodic Sequence
1	1, 2, 4, 8, 16, 27, 13, 26, 15, 30, 7, 14, 28, 3, 6, 12, 24, 11, 22, 23, 21, 17, 25, 9, 18, 31, 5, 10, 20, 19, 29

LIST OF REFERENCES

1. Breuer, M. A., and Carlan, A. J., *State of the Art Assessement of Testing and Testability of LSI/VLSI Circuits,* pp. 10-11, Engineering Group Report, October 1982.

2. Weste, N., and Eshraghian, K., *Principles of CMOS VLSI Design, A System Persepective,* Addision-Wesley, 1985.

3. Koenemann, B., Mucha, J., and Zwiehoff, G., *Built-in Logic Block Observation Techniques,* pp. 37-41, International Test Conference, October, 1977.

4. Hewlett, P. J., *Signature Analysis,* Vol. 28, n. 9, May, 1977.

5. Barus, J., *An Analysis of Aliasing Probability in BIST,* June 1991.

6. Tsui, F. F., *LSI/VLSI Testability Design,* pp. 173-175, Mcgraw-Hill Inc., 1987.

7. Parker, K. P., *Adaptive Random Test Generation,* Vol. 1, pp. 62-68, J. Design Ant, 1976.

8. Siavohi, F., *A Novel Weighted Test-pattern Generation Approach for VLSI Built-in Self Test,* pp. 256-257, International Test Conference, 1988.

9. Timoc, C., Stott, F., and Hess, L., *Adaptive Self Test for Microprocessor,* pp. 701-703, Proceedings of International Test Conference, 1983.

10. Tanenbaum, A. S., *Structured Computer Organization,* pp. 96-98, Prentice-Hall Inc., 1990.

11. Bardell, P. H., *Built-in Test for VLSI,* pp. 10-11, Wiley-Interscience Publication, 1987.

12. *VLSI Design Tools References Manual Release 3.2,* pp. 3-5, N.W Laboratry for Integrated Systems, September, 1988.

13. Waicukauski, J. A., and Lindbloom, E., *Fault Detection Effectiveness of Weighted Random Patterns,* pp. 245-246, International Test Conference, 1988.

14. Schnurmann, H. D., Lindbloom, E., and Carpenter, R. G., *The Weighted Random Test-Pattern Generator*, pp. 695-700, IEEE Transactions on Computers, 1975.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2. Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3. Department Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4. Prof. Chyan Yang, Code EC/Ya Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	3
5. Prof. Herschel H. Loomis, Jr., Code EC/Lm Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	3
6. Chin, Miao 4th Flr, No. 12, Alley 32, Lane 66 Wan Te Road, Nei Hu Taipei Taiwan, Republic of China	1
7. T.F.P.G Library P.O. Box 8761 Ta-Fu, I-Lan Taiwan, Republic of China	1
8. Library of Chinese Naval Academy P.O. Box 8494 Tsoying Kaohsiung, Taiwan, Republic of China	2

Thesis
C4445 Chin
c.1 Complementary metal oxide silicon cyclic redundancy check generators.